Incremental incomplete LU factorizations with applications

Caterina Calgaro, Jean-Paul Chehab, Yousef Saad

NUMERICAL LINEAR ALGEBRA WITH APPLICATIONS
Numer. Linear Algebra Appl. 2009; **0**:1 *Prepared using* nlaauth.cls *[Version: 2002/09/18 v1.02]*

Incremental Incomplete LU factorizations with applications

Caterina Calgaro[1], Jean-Paul Chehab[2] and Yousef Saad[3,*]

[1] *Université Lille 1, Laboratoire Paul Painlevé, UMR 8524, France and INRIA Lille Nord Europe, EPI SIMPAF, France.* e-mail:caterina.calgaro@univ-lille1.fr
[2] *Université de Picardie Jules Verne, LAMFA, UMR 6140, Amiens, France and INRIA Lille Nord Europe, EPI SIMPAF, France.* e-mail:jean-paul.chehab@u-picardie.fr
[3] *University of Minnesota, Computer Science, USA.* e-mail: saad@cs.umn.edu †

SUMMARY

This paper addresses the problem of computing preconditioners for solving linear systems of equations with a sequence of slowly varying matrices. This problem arises in many important applications. For example, a common situation in computational fluid dynamics, is when the equations change only slightly, possibly in some parts of the physical domain. In such situations it is wasteful to recompute entirely any LU or ILU factorizations computed for the previous coefficient matrix. A number of techniques for computing incremental ILU factorizations are examined. For example we consider methods based on approximate inverses as well as alternating techniques for updating the factors L and U of the factorization. Copyright © 2009 John Wiley & Sons, Ltd.

KEY WORDS: Preconditioning, Incomplete LU factorization, incremental LU.

1. Introduction

When solving large sparse linear systems of equations, it is generally accepted that preconditioned Krylov subspace methods are effective alternatives to direct solution methods which offer a good compromise between efficiency and robustness. These methods combine an accelerator such as GMRES or Bi-CGSTAB and a preconditioner such the ILU factorization with threshold or with level-of-fill, see, e.g., [28, 1] for details. Here, a few techniques are presented for constructing ILU-type preconditioners in an incremental manner, i.e., starting from an already computed approximate factorization. There are many reasons why such techniques may be desirable.

For example, a common problem which arises in many complex applications is to solve a sequence of linear systems of the form:

$$A_k x_k = b_k, \tag{1}$$

*Correspondence to: e-mail: saad@cs.umn.edu
†Work initiated while the third author was visiting the department of Mathematics of the University of Lille. Work of the third author was supported by the US Department Of Energy under contract DE-FG-08ER25841 and by the Minnesota Supercomputer Institute.

Received
Revised

for $k = 1, 2, ...$ In these applications A_k does not generally change too much from one step k to the next, as it is often the result of a continuous process (e.g. A_k can represent the discretization of some problem at time t_k). The problem then is to solve each consecutive system effectively, by taking advantage of earlier systems if possible. This problem arises for example in computational fluid dynamics, when the equations change only slightly possibly in some parts of the domain. A similar situation was recently considered in [5].

There are many possible other applications of incremental ILU factorizations. One that is often neglected concerns the construction of preconditioners progressively within the whole iterative cycle. Thus, when solving a linear system we do not know in advance how accurate the preconditioner must be for the iteration to converge fast enough. One could then start with a rough factorization which is gradually improved within a Krylov subspace acceleration. As soon as there are signs of stagnation – or slow convergence – one can attempt to improve the current factors by running one step of improvement with the help of one of several procedures that will be introduced in this paper.

It is also possible to devise strategies to build ILU factorizations in this way. Starting with the SSOR(1) factorization, one can compute successive improvements until a satisfactory factorization is obtained. As will be seen, a by-product of one of the techniques introduced is that one can compute an exact LU factorization by a sort if alternating procedure. We will establish the surprising result that this iterative procedure in fact converges in N steps, where N is the matrix size.

A somewhat related problem which was examined in the past is that of updating the (exact) LU factorization of a given matrix when a very small number of its entries change. This particular case received much attention in the power systems literature, see, e.g., [9]. In contrast, the problem of updating an existing ILU factorization has received little attention so far. Birken et al. [5], consider this problem specifically for CFD equations and proposed an inexpensive approach to update one of the factors whey they undergo small changes. The basic idea of their method is quite simple and is discussed in Section 4.3.

Updating preconditioners is quite easy when approximate inverse preconditioners are used because these methods typically involve explicit updates. We will start by exploring techniques for improving incomplete LU factorizations by exploiting approximate inverse strategies. We will then discuss a set of techniques based on alternating procedures for correcting an ILU factorization.

In the remainder of this section we state the specific problem addressed in this paper, and outline in broad terms the two classes of methods considered for solving it. Some notation which will be used throughout the paper will also be introduced.

1.1. The problem

Suppose we have an approximate factorization which we write in the form

$$A = LU + R, \tag{2}$$

where R is the error in the factorization, which corresponds to the matrix of the terms that are dropped during the factorization procedure. Our primary goal is to improve this factorization, i.e., to find sparse matrices X_L, X_U such that $L + X_L$, $U + X_U$ is a better pair of factors than L, U. Consider the new error $A - LU$ for the new factors.

$$A - (L + X_L)(U + X_U) = (A - LU) - X_L U - L X_U - X_L X_U \ . \tag{3}$$

Ideally, we would like to make the right-hand side equal to zero. Denoting by R the matrix $A - LU$, this means we would like to solve:

$$X_L U + L X_U + X_L X_U - R = 0 \ , \tag{4}$$

for X_L and X_U. The above equation, which can be viewed as a form of Riccati equation, is nonlinear in the pair of unknowns X_L, X_U. In the symmetric case, where LU is replaced by the Cholesky factorization, and where we set $X_U = X_L^T \equiv X$ by symmetry, then (4) becomes

$$X L^T + L X^T + X X^T - R = 0 \ , \tag{5}$$

the standard Algebraic Riccati Equation.

In our case, we only consider (4) which we would like to solve approximately. For this we begin by neglecting the quadratic term $X_L X_U$, leading to:

$$X_L U + L X_U - R = 0 \ . \tag{6}$$

We will consider two broad paths to tackle the above problem. First, we will consider a few approaches in the spirit of approximate inverse-type techniques which will try to find sparse triangular factors X_L, X_U that approximately minimize the Frobenius norm of the left-hand side of (6). Second, we exploit an alternating procedure at the matrix level which fixes U (i.e., we set $X_U = 0$) and solves for X_L, and then fixes the resulting L and solve for X_U, and repeat the process. It is also possible to use a similar alternating procedure but at the vector level – where columns of L and rows of U are updated alternatively. This approach, which is similar to a well-known one used for solving Lyapunov equations, is not considered here. Similarly, methods based on differential equations similar to those discussed in [11, 10] can also be of interest in this context but will not be considered in this paper.

1.2. Notation

We will often need to extract lower and upper triangular parts of matrices. Given an $N \times N$ matrix X, we denote by $X_{\backslash}$ the *strict lower triangular part* of X, and by $X_{\not\backslash}$ the *upper triangular part* of X (which includes the diagonal of X). The *unit lower triangular matrix* obtained by taking the lower triangular part of X and replacing its diagonal entries by ones, is denoted by $X_{\not\backslash}$. In other words $X_{\not\backslash} = I + X_{\backslash}$, the identity matrix plus the *strict* lower part of X. Note the following easy to verify properties of these operators

$$(X+Y)_{\backslash} = X_{\backslash} + Y_{\backslash}; \qquad (X+Y)_{\not\backslash} = (X+Y)_{\not\backslash}; \tag{7}$$

$$(X_{\not\backslash} Y_{\backslash})_{\not\backslash} = 0; \qquad (X_{\backslash} Y_{\not\backslash})_{\not\backslash} = 0; \tag{8}$$

$$(X_{\not\backslash} Y_{\not\backslash})_{\backslash} = 0 \ . \tag{9}$$

The fact that a matrix is upper triangular can be conveniently expressed by the equality $X_{\not\backslash} = X$. Similarly, a matrix is strict lower triangular when $X_{\backslash} = X$ and is unit lower triangular when $X_{\not\backslash} = X$. This notation is specifically designed for the LU factorization which involves unit lower triangular and upper triangular matrices.

The inner product of two $N \times N$ matrices, is defined as:

$$\langle X, Y \rangle = \mathrm{Tr}(Y^T X). \tag{10}$$

The Frobenius norm $\|.\|_F$ is the 2-norm associated with this inner product, i.e., $\|X\|_F = [\mathrm{Tr}(X^T X)]^{1/2}$.

2. Techniques based on approximate inverses

Standard approximate inverse techniques attempt to find an approximate inverse to A which is sparse, see, e.g., [3, 2, 4, 26, 24, 25, 20, 21]. One such approach used in APINV (see, e.g., [13]) is to approximately minimize the Frobenius norm,

$$\|I - AM\|_F \tag{11}$$

over sparse matrices M which have a pre-specified pattern. We are interested in a by-product of this general class of methods which we term "Sparse Matrix Corrections".

2.1. Sparse Matrix Corrections

If B is a given preconditioner to A, we can seek a matrix M such that AM approximates B instead of the identity matrix as is done in (11). The approximation can be sought in the Frobenius norm sense. More precisely, let $B = LU$ be a given ILU preconditioner of A. We can seek a matrix M such that AM approximates $B = LU$ by trying to approximately minimize $\|B - AM\|_F$. Note that $B = LU$ is generally a sparse matrix. The corrected preconditioning operation would then become $MB^{-1} = MU^{-1}L^{-1}$. This approach was first advocated in [13] (see also [28, sec. 10.5.8]), and was also exploited in [12]. It was later discussed in [23] where the term "target matrices" was introduced for the matrix B. In the above equation M can be obtained by approximately solving the linear systems $Am_j = b_j$, where b_j, the j-th column of B is sparse, in order to find the j-th (sparse) column m_j of M.

An alternative to solving individual successive linear systems $Am_j = e_j$, is to seek a global iteration, i.e., an iteration which updates the whole matrix at once as in

$$M_{new} = M + \alpha G \tag{12}$$

where G, the 'search direction', is a certain matrix. It is possible to apply a form of the steepest descent algorithm to try to minimize

$$F(M) = \|B - AM\|_F^2, \tag{13}$$

which is a quadratic function on the space of $N \times N$ matrices, viewed as objects in $\mathbb{R}^{N^2}$. This requires the gradient of the objective function of $F(M)$ in (13). The natural inner product on the space of matrices, with which the function (13) is associated, is the inner product (10).

In descent-type algorithms the scalar α in (12) is selected to minimize the objective function associated with M_{new}. The simplest choice for the descent direction G is to take it to be the residual matrix $R = B - AM$, where M is the new iterate. The corresponding descent algorithm is called the "Minimal Residual (MR)" algorithm. It is easy to see that R is indeed a descent direction for F. See [13] for details on the resulting global Minimal Residual algorithm.

Another popular choice is to take G to be the direction of steepest descent, i.e., the direction opposite to the gradient of F. A result of [13] can be expanded to show that *the array representation of the gradient of F with respect to M is the matrix $G = -2A^T R$*. As noted in [13] this option can lead to a very slow convergence in some cases because it is essentially a steepest descent-type algorithm applied to the normal equations. In either global steepest descent or minimal residual, we need to form and store the G matrix explicitly.

The tools just discussed can be adapted to develop algorithms for improving a given ILU factorization, by alternatively correcting the L and U factors in turn.

2.2. Alternating L-U descent methods

Consider first equation (4) with the particular choice $X_L = 0$, which corresponds to updating U while L is kept frozen. Then, a sparse X_U is sought which minimizes,

$$F(X_U) = \|A - L(U + X_U)\|_F^2 = \|R - LX_U\|_F^2 \tag{14}$$

where we set $R \equiv A - LU$, the current factorization error.

Similar to the situation of approximate inverses, the optimum X_U is trivially known, in this case it equals $L^{-1}R$, and this will be exploited in the next section. Here, we seek an approximation only to this exact solution. Note that (14) is of the same form as (13). We have

$$\|R - LX_U\|_F^2 = \text{Tr } ([R - LX_U]^T[R - LX_U]) = \|R\|_F^2 - 2\text{Tr}(R^T LX_U) + \|LX_U\|_F^2 \ .$$

Therefore, the gradient of $F(X_U)$ at $X_U = 0$ written in matrix form is simply $G = -2\ L^T R$. What this means is that when X_U is a small multiple of $L^T R$ then $F(X_U)$ will decrease. With this, a steepest descent method can now be devised. The only issue is that $G = L^T R$ is not necessarily upper triangular, so X_U which is a multiple of G in this situation will not be upper triangular as desired. We can define G to be the upper triangular part of $L^T R$, i.e., using the notation defined in Section 1.2,

$$G = [L^T R]_{\mathbb{U}}, \tag{15}$$

which represents the upper triangular part of $L^T R$ (including the diagonal). We may need to sparsify G matrix even further to limit fill-in. Remarkably, the resulting matrix remains a descent direction under these changes. Here descent direction is to be interpreted as to mean 'non-ascent'.

Lemma 1. *The matrix $G = [L^T R]_{\mathbb{U}}$ and any sparsified version $\tilde{G}$ of it is a descent direction for F at $X_U = 0$.*

Proof. Let us denote by X the matrix $L^T R$. Since the gradient is $-2X$, a matrix G is a descent direction if $< -2X, G > \leq 0$, or $< X, G >= \text{Tr}(G^T X) \geq 0$. Let η_{ij} be the entries of X and consider any diagonal entry of $G^T X$, where G is the upper triangular part of X. We have:

$$[G^T X]_{jj} = \sum_{i=1}^{j} \eta_{ij}^2 \geq 0 \ .$$

Therefore the trace is ≥ 0. If G is further sparsified to $\tilde{G}$, then the above sum is replaced by a sum over the sparsity pattern of the j-th row of G^T, i.e., the sparsified pattern of the j-th column of G, and it remains non-negative. ∎

To determine the best α in (12) we must minimize the quadratic form:

$$\|R - \alpha LG\|_F^2 = \text{Tr } ([R - \alpha LG]^T[R - \alpha LG]) = \|R\|_F^2 - 2\alpha\text{Tr}(R^T LG) + \alpha^2\|LG\|_F^2,$$

yielding

$$\alpha_U = \frac{\langle R, LG\rangle}{\langle LG, LG\rangle} = \frac{\text{Tr}(R^T LG)}{\text{Tr}\,((LG)^T LG)}. \tag{16}$$

Formulas for the L part can be readily obtained by repeating the above argument, replacing the objective function (14) by

$$\|A - (L + X_L)U\|_F^2 = \|R - X_L U\|_F^2 \ .$$

The array representation of the gradient becomes $-2RU^T$. We seek a correction X_L to L that is strict lower triangular, so we define $G_L = [RU^T]_{\mathbb{L}}$ and

$$\alpha_L = \frac{\langle R, G_L U\rangle}{\langle G_L U, G_L U\rangle} = \frac{\mathrm{Tr}(R^T G_L U)}{\mathrm{Tr}\left((G_L U)^T G_L U\right)}. \tag{17}$$

Note that the denominators in (16) and (17) may be computed as $\|LG\|_F^2$ and $\|G_L U\|_F^2$ respectively. An alternating algorithm which updates U and L in turn, can now be sketched.

Algorithm 1 (MERLU, Minimal Energy Residual descent for LU)

1. Select an initial pair L, U (with $L = L_{\mathbb{L}}$, $U = U_{\mathbb{U}}$)
2. Until convergence Do
3. Compute $R := A - LU$
4. Compute $G = [L^T R]_{\mathbb{U}}$
5. Apply numerical dropping to G
6. Compute $\alpha = \langle R, C\rangle / \|C\|_F^2$, where $C = LG$.
7. Compute $U := U + \alpha G$
8. Compute $R := A - LU$
9. Compute $G := [RU^T]_{\mathbb{L}}$
10. Apply numerical dropping to G
11. Compute $\alpha = \langle R, C\rangle / \|C\|_F^2$ where $C = GU$.
12. Compute $L := L + \alpha G$
13. EndDo

With the above approach it is guaranteed that the residual norm $\|R\|_F$ will not increase. A convenient feature of the method is that it can tolerate dropping as indicated by Lemma 1. One possible practical concern is the cost of this procedure. If we perform one iteration to improve a given preconditioner, the computation will be dominated by the formation of the gradient matrices G in Lines 4 and 9 and the calculation of the scalars α in lines 6 and 11. These involve the products of two sparse matrices. Note that the matrix C need not be stored. It can be calculated one column at a time, for example. Then the norm of the column and the inner product of this column with the corresponding column of R can be evaluated to compute parts of $\langle R, C\rangle$ and $\langle C, C\rangle$ and then the column can be discarded.

3. Alternating Sparse-Sparse iterations

Consider equation (4) in which we set $X_U = 0$. If U is nonsingular we obtain:

$$X_L U = R \quad \rightarrow \quad X_L = RU^{-1}. \tag{18}$$

Thus, the correction to L can be obtained by solving a sparse triangular linear system with a sparse right-hand side matrix, i.e., the system $U^T X_L^T = R^T$. However, as was noted before, the updated matrix $L + X_L$ obtained in this way is not necessarily unit lower triangular and it is therefore required to apply the $[\ .\ ]_{\mathbb{U}}$ operation for $L + X_L$ or the $[\ .\ ]_{\mathbb{L}}$ operation for X_L.

This procedure can be repeated by freezing U and updating L and vice-versa alternatingly, leading to an iteration which can be summarized by the following two equations:

$$U_{k+1} = U_k + [L_k^{-1}(A - L_k U_k)]_{\mathbb{U}}, \quad (19)$$
$$L_{k+1} = L_k + [(A - L_k U_{k+1})U_{k+1}^{-1}]_{\mathbb{L}}. \quad (20)$$

A feature of equations (19–20) is that they simplify. For (19) for example, we have

$$U_{k+1} = U_k + [L_k^{-1}(A - L_k U_k)]_{\mathbb{U}} = [U_k + L_k^{-1}(A - L_k U_k)]_{\mathbb{U}} = [L_k^{-1}A]_{\mathbb{U}} \ .$$

A similar expression for L_{k+1} is easily derived and this leads to an iteration of the form:

$$U_{k+1} = [L_k^{-1}A]_{\mathbb{U}}, \quad (21)$$
$$L_{k+1} = [AU_{k+1}^{-1}]_{\mathbb{L}}. \quad (22)$$

Note that *the above two formulas are not intended for practical use.* It is generally more appealing to use (19–20) because one can sparsify $A - L_k U_k$ (say) keeping only the largest entries and making the solve much less inexpensive.

One issue in the above equations is the fact that the algorithm will break down when U_{k+1} is singular, and this may indeed happen, as the following example shows.

Example

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 4 \end{pmatrix} \rightarrow L_0 = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -1 & 1 \end{pmatrix} \rightarrow L_0^{-1}A = \begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 0 \\ 3 & 5 & 7 \end{pmatrix} \rightarrow U_1 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 7 \end{pmatrix}.$$

This issue is not too different from that encountered in incomplete LU factorizations with threshold. In fact any form of LU factorization implicitly involves solves with lower and upper triangular matrices and when dropping is applied, these matrices can become singular. The occurrence of this breakdown is rare in practice. Note that at the limit, U_k will approach the U factor of the LU factorization of A (as will be shown in Proposition 1), a nonsingular matrix. The resulting alternating algorithm is given below.

Algorithm 2 (ITALU, Iterative Threshold Alternating Lower-Upper correction)

1. Given: A, U_0, L_0 (with $U_0 = [U_0]_{\mathbb{U}}; L_0 = [L_0]_{\mathbb{L}}$)
2. For $k = 0, \cdots$, Do:
3. Compute $R_k = A - L_k U_k$
4. Compute $X_U = [L_k^{-1} R_k]_{\mathbb{U}}$
5. Apply numerical dropping to X_U
6. $U_{k+1} = U_k + X_U$
7. If $det(U_{k+1}) == 0$ Abort "Singular U reached"
8. Compute $R_{k+1/2} = A - L_k U_{k+1}$
9. Compute $X_L = [R_{k+1/2} U_{k+1}^{-1}]_{\mathbb{L}}$
10. Apply numerical dropping to X_L
11. $L_{k+1} = L_k + X_L$
12. EndDo

It may be thought that the triangular systems in Lines 4 and 9 of the algorithm are expensive to solve. However, one must remember that the right-hand side is sparse and there exist effective techniques for this task. This is discussed in detail in Section 4. A variation to the dropping steps in Lines 5 and 10, would be to drop terms from the residual matrices R_k and $R_{k+1/2}$ instead of the matrices X_U and X_L.

Expressions (21–22) can be further simplified by replacing A by one of its triangular parts. This is a result of the following simple lemma which will also be helpful in the next section.

Lemma 2. *Let X be any square matrix and let P be a unit lower triangular matrix and Q an upper triangular matrix. Then the following equalities hold:*

$$[PX]_{\mathbb{U}} = [PX_{\mathbb{U}}]_{\mathbb{U}} \quad ; \quad [XQ]_{\mathbb{L}} = [X_{\mathbb{L}}Q]_{\mathbb{L}} \ . \tag{23}$$

Proof. The first relation results from the equalities (7) and (8). Indeed,

$$[PX]_{\mathbb{U}} = [P(X_{\mathbb{L}} + X_{\mathbb{U}})]_{\mathbb{U}} = [PX_{\mathbb{L}}]_{\mathbb{U}} + [PX_{\mathbb{U}})]_{\mathbb{U}} = 0 + [PX_{\mathbb{U}})]_{\mathbb{U}}.$$

The second equality can be shown similarly from (7) and (9):

$$[XQ]_{\mathbb{L}} = [(X_{\mathbb{L}} + X_{\mathbb{U}})Q]_{\mathbb{L}} = [X_{\mathbb{L}}Q]_{\mathbb{L}} + [X_{\mathbb{U}}Q]_{\mathbb{L}} = [X_{\mathbb{L}}Q]_{\mathbb{L}} + 0.$$

■

Note that other similar equalities can also be shown. For example we will later exploit the relation: $[XQ]_{\mathbb{L}} = [X_{\mathbb{L}}Q]_{\mathbb{L}}$.

Using the lemma with $P = L_k^{-1}$, $Q = U_{k+1}^{-1}$, and $X = A$, equations (21–22) become:

$$U_{k+1} = [L_k^{-1}A_{\mathbb{U}}]_{\mathbb{U}}, \tag{24}$$

$$L_{k+1} = [A_{\mathbb{L}}U_{k+1}^{-1}]_{\mathbb{L}}. \tag{25}$$

3.1. Convergence results in the dense filter case

Filtering consists of keeping only specific wanted entries in a matrix, and dropping the others, i.e., replacing them with zeros. For example, applying an upper triangular filter to a matrix M amounts to only keeping its upper triangular part. An equivalent term often used in the literature is that of a 'mask'. The filter matrix can be dense or sparse. Next, we establish a few properties of the sequences U_k and L_k in the particular case where we use *dense triangular filters*. This only means that the iterates L_k, U_k are evaluated by the expressions (21–22) exactly (no other dropping is performed). We start with the following lemma.

Lemma 3. *Let two square matrices P and Q be unit lower triangular and upper triangular respectively, with Q nonsingular. Then for any square matrix M we have:*

$$[M]_{\mathbb{U}} = [[MQ^{-1}]_{\mathbb{U}}Q]_{\mathbb{U}} \quad \textit{and} \quad [M]_{\mathbb{U}} = [P[P^{-1}M]_{\mathbb{U}}]_{\mathbb{U}}. \tag{26}$$

Proof. The second part of the equation in (23) applied to $X = MQ^{-1}$ together with the definition $[.]_{\mathbb{U}} = I + [.]_{\mathbb{L}}$, yields immediately

$$[[MQ^{-1}]_{\mathbb{U}}Q]_{\mathbb{U}} = [(MQ^{-1})Q]_{\mathbb{U}} = [M]_{\mathbb{U}}.$$

The second relation in (26) follows similarly by taking $X = P^{-1}M$ in the first equality of equation (23) ■

We can now prove the following proposition

Proposition 1. *Let A be a matrix that admits the LU decomposition $A = LU$, where L is unit lower triangular and U is upper triangular. Then, the sequences U_k and L_k defined by (21) and (22) starting with an arbitrary (unit lower triangular) L_0, satisfy the following properties*

(i) $[A - L_k U_{k+1}]_{\mathbb{\downarrow}} = 0, \quad [A - L_{k+1} U_{k+1}]_{\mathbb{\downarrow}} = 0,\ k \geq 0.$

(ii) *If the sequences L_k and U_k converge, then $\lim_{k\to+\infty} L_k = L$ and $\lim_{k\to+\infty} U_k = U$.*

(iii) *The algorithm converges in one step if $L_0 = L$.*

Proof. We first prove assertion (i). From (21) it follows that $L_k U_{k+1} = L_k [L_k^{-1} A]_{\mathbb{\downarrow}}$. Applying an upper triangular filter to both sides, we obtain, thanks to the second part of Equation (26)

$$[L_k U_{k+1}]_{\mathbb{\downarrow}} = [A]_{\mathbb{\downarrow}}.$$

This proves the first half of (i). To prove the second half first observe that from (23) we have $[L_{k+1} U_{k+1}]_{\mathbb{\downarrow}} = [(L_{k+1})_{\mathbb{\downarrow}} U_{k+1}]_{\mathbb{\downarrow}}$. Then, using the relation $L_{k+1} = [A U_{k+1}^{-1}]_{\mathbb{\Downarrow}}$, and invoking Equation (26) once more yields:

$$[L_{k+1} U_{k+1}]_{\mathbb{\downarrow}} = [[A U_{k+1}^{-1}]_{\mathbb{\Downarrow}} U_{k+1}]_{\mathbb{\downarrow}} = [A]_{\mathbb{\downarrow}}.$$

To prove assertion (ii), we introduce $\bar{L} = \lim_{k\to+\infty} L_k$ and $\bar{U} = \lim_{k\to+\infty} U_k$. Taking limits in (i) yields the relations

$$[\bar{L}\bar{U}]_{\mathbb{\downarrow}} = [A]_{\mathbb{\downarrow}} \quad \text{and} \quad [\bar{L}\bar{U}]_{\mathbb{\downarrow}} = [A]_{\mathbb{\downarrow}}\ .$$

Hence $\bar{L}\bar{U} = A$ and the uniqueness of the LU factorization implies that $\bar{L} = L$ and $\bar{U} = U$. Finally, when we take $L_0 = L$, then $U_1 = [L^{-1} L U]_{\mathbb{\downarrow}} = [U]_{\mathbb{\downarrow}} = U$, proving (iii). ∎

In fact, if one of the sequences L_k or U_k converges, then both of them will converge to their limits L and U respectively. This follows from the following relations

$$\begin{aligned} U_{k+1} - U &= [L_k^{-1} A - U]_{\mathbb{\downarrow}} = [L_k^{-1} A - L^{-1} A]_{\mathbb{\downarrow}} = [(L_k^{-1} - L^{-1}) A]_{\mathbb{\downarrow}} \\ L_{k+1} - L &= [A U_{k+1}^{-1} - L]_{\mathbb{\downarrow}} = [A U_{k+1}^{-1} - A U^{-1}]_{\mathbb{\downarrow}} = [A(U_{k+1}^{-1} - U^{-1})]_{\mathbb{\downarrow}}. \end{aligned}$$

These two relations show that as long as there is no break-down (i.e., U_{k+1} is nonsingular each time it is computed from (24)) the sequences L_k, U_k are well defined, and they converge hand-in hand, in the sense that when L_k converges, then U_k also converges and vice versa.

We can now prove the following convergence result which establishes the surprising fact that under the condition that the filters are triangular and dense, the sequence of L_k's and U_k's terminates in a finite number of steps.

Theorem 1. *Let an initial L_0 be given such that the sequences L_k, U_k are defined for $k = 1, 2, \cdots, N$. Then the sequences U_k and L_k converge to U and L respectively, in no more than N steps.*

Proof. We proceed by induction on the dimension N of the matrices. The result is trivial to establish for the case $N = 1$. Indeed, if $A = [a]$, the trivial LU factorization is $L = [1]$, $U = [a]$ and the sequence L_k, U_k reaches this factorization in one step.

We now assume that the algorithm converges in no more than $N - 1$ steps for matrices of dimension $N - 1$. We use the notation

$$A = \begin{pmatrix} \tilde{A} & b \\ c & d \end{pmatrix}, \quad L_k = \begin{pmatrix} \tilde{L}_k & 0 \\ \alpha_k & 1 \end{pmatrix}, \quad U_k = \begin{pmatrix} \tilde{U}_k & \gamma_k \\ 0 & \delta_k \end{pmatrix}.$$

We also consider the block LU factorization of A,

$$A = \underbrace{\begin{pmatrix} \tilde{L} & 0 \\ c\,\tilde{U}^{-1} & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} \tilde{U} & \tilde{L}^{-1}b \\ 0 & \delta \end{pmatrix}}_{U} \quad \text{with} \quad \delta = d - c\tilde{U}^{-1}\tilde{L}^{-1}b. \tag{27}$$

The recurrence relations (21–22) of the algorithm give,

$$U_{k+1} = \begin{pmatrix} \tilde{U}_{k+1} & \gamma_{k+1} \\ 0 & \delta_{k+1} \end{pmatrix} = \begin{pmatrix} [\tilde{L}_k^{-1}\tilde{A}]_{\nabla} & \tilde{L}_k^{-1}b \\ 0 & -\alpha_k \tilde{L}_k^{-1}b + d \end{pmatrix}. \tag{28}$$

and

$$L_{k+1} = \begin{pmatrix} \tilde{L}_{k+1} & 0 \\ \alpha_{k+1} & 1 \end{pmatrix} = \begin{pmatrix} [\tilde{A}\tilde{U}_{k+1}^{-1}]_{\mathbb{L}} & 0 \\ c\tilde{U}_{k+1}^{-1} & 1 \end{pmatrix}. \tag{29}$$

This results in the relations

$$\tilde{U}_{k+1} = [\tilde{L}_k^{-1}\tilde{A}]_{\nabla}, \quad \tilde{L}_{k+1} = [\tilde{A}\tilde{U}_{k+1}^{-1}]_{\mathbb{L}}, \tag{30}$$

and

$$\alpha_{k+1} = c\tilde{U}_{k+1}^{-1}, \quad \gamma_{k+1} = \tilde{L}_k^{-1}b, \quad \delta_{k+1} = d - \alpha_k \tilde{L}_k^{-1}b. \tag{31}$$

Now by assumption, the relations (30) show that the sequence of the pairs $\tilde{L}_k, \tilde{U}_k$ converges in no more than $N-1$ steps to the L, U factors of $\tilde{A}$, i.e., $\tilde{L}_{N-1} = \tilde{L}$ and $\tilde{U}_{N-1} = \tilde{U}$. It follows that $\tilde{L}_N = \tilde{L}$, $\tilde{U}_N = \tilde{U}$, $\alpha_N = c\tilde{U}^{-1}$, $\gamma_N = \tilde{L}^{-1}b$, $\delta_N = d - c\tilde{U}^{-1}\tilde{L}^{-1}b$. Taking $k = N-1$ in (28) and comparing the result with the U part of the factorization (27) we obtain immediately $U_N = U$. Proceeding the same way by taking $k = N-1$ in (29) and comparing the result with the L factor in (27) shows that $L_N = L$ and completes the proof. ∎

It is interesting to observe that the above theorem provides an alternative way of computing the (exact) LU factorization of a matrix. Like Gaussian elimination, it is a direct method, though a costlier one in the dense case. Specifically, the full N steps of the algorithm require $O(N^4)$ operations to complete in the dense case. However, in general the algorithm is likely to converge in fewer than N steps.

3.2. Convergence in the sparse filter case

Consider now the situation where a general sparse filter is applied. This means that a sparsity pattern $\mathcal{F}_k$ is selected at each step k and the matrices L_k, U_k of the factorization are restricted to having the pattern of the lower and upper parts of $\mathcal{F}_k$, respectively, by dropping any entries outside of this pattern. Recall the notation with filters. The filter $\mathcal{F}$ can be considered as an $N \times N$ matrix of zeros and ones. A zero entry corresponds to an unwanted term in the pattern, so a nonzero entry in this location is dropped. We denote by $X \odot \mathcal{F}$ the Hadamard product (i.e., component-wise product), of the matrices X and $\mathcal{F}$. The algorithm with a general constant sparse filter would replace equations (21–22) by

$$U_{k+1} = [(L_k^{-1}A) \odot \mathcal{F}]_{\nabla} = [(L_k^{-1}A)]_{\nabla} \odot \mathcal{F} \tag{32}$$

$$L_{k+1} = [(AU_{k+1}^{-1}) \odot \mathcal{F}]_{\mathbb{L}} = [(AU_{k+1}^{-1})]_{\mathbb{L}} \odot \mathcal{F}\ . \tag{33}$$

The algorithm breaks down immediately if the filter has a zero diagonal entry. These *singular* filters and excluded from consideration, i.e., all filters considered are such that $\mathcal{F}_{ii} = 1,\ \forall i$.

A desirable extension of the convergence result of the previous section would be to show that the sequences L_k, U_k converge to the sparse LU factors L and U of A respectively. For this to be possible, the patterns of the filter matrices $\mathcal{F}_k$ must contain those of the L and U parts respectively at each step. In fact under the assumption that $\mathcal{F}_k$ contains the symbolic pattern of Gaussian elimination, the same result as that of the previous theorem holds. Here, by symbolic pattern, we mean the pattern of Gaussian elimination obtained from the graph, i.e., in the situation where exact cancellations do not take place in Gaussian elimination [14]. The proof of this result is an immediate extension of that of the previous theorem. The result is stated without proof as a corollary.

Corollary 1. *Assume that each of the filter matrices $\mathcal{F}_k$, $k \geq 0$ contains the (symbolic) sparsity patterns of L and U respectively and let an initial L_0 be given such that $L_0 \odot \mathcal{F}_0 = L_0$ and the sequences L_k and U_k exist for $k = 1, \cdots, N$. Then the sequences U_k and L_k converge to U and L, respectively, in no more than N steps.*

It is interesting to consider how the results of Proposition 1 change when a general sparse filter is applied. Consider the iteration (32), (33), in which the filter $\mathcal{F}$ is now constant. We can rewrite the iteration with the help of the upper and lower parts of the filter as follows:

$$U_{k+1} = [(L_k^{-1}A)]_{\mathbb{U}} \odot \mathcal{F}_{\mathbb{U}}, \tag{34}$$

$$L_{k+1} = [(AU_{k+1}^{-1})]_{\mathbb{L}} \odot \mathcal{F}_{\mathbb{L}}. \tag{35}$$

The filtering operation in (34) is equivalent to subtracting an upper triangular sparse matrix S_k from the matrix $(L_k^{-1}A)_{\mathbb{U}}$, so it follows that $U_{k+1} = [(L_k^{-1}A)]_{\mathbb{U}} - S_k$. Proceeding as in Proposition 1 we multiply this relation to the left by L_k, so $L_kU_{k+1} = L_k[L_k^{-1}A]_{\mathbb{U}} - L_kS_k$, and then apply the upper triangular filter to obtain, exploiting again Lemma 3,

$$[L_kU_{k+1}]_{\mathbb{U}} = [L_k(L_k^{-1}A)_{\mathbb{U}}]_{\mathbb{U}} - [L_kS_k]_{\mathbb{U}} = [A]_{\mathbb{U}} - [L_kS_k]_{\mathbb{U}}. \tag{36}$$

Similarly, equation (35) yields $L_{k+1} = [(AU_{k+1}^{-1})]_{\mathbb{L}} \odot \mathcal{F}_{\mathbb{L}} = [(AU_{k+1}^{-1})]_{\mathbb{L}} - T_k$, where T_k represents the (sparse) lower triangular matrix of dropped entries. Then, proceeding in the same way as above, we get $L_{k+1}U_{k+1} = [(AU_{k+1}^{-1})]_{\mathbb{L}}U_{k+1} - T_kU_{k+1}$ and

$$[L_{k+1}U_{k+1}]_{\mathbb{L}} = [[(AU_{k+1}^{-1})]_{\mathbb{L}}U_{k+1}]_{\mathbb{L}} - [T_kU_{k+1}]_{\mathbb{L}} = A_{\mathbb{L}} - [T_kU_{k+1}]_{\mathbb{L}}. \tag{37}$$

We have just shown the following two relations

$$[A - L_kU_{k+1}]_{\mathbb{U}}\,, = [L_kS_k]_{\mathbb{U}} \tag{38}$$

$$[A - L_{k+1}U_{k+1}]_{\mathbb{L}}\,. = [T_kU_{k+1}]_{\mathbb{L}} \tag{39}$$

Though these relations do not establish convergence, they tell us something in the case when the process converges.

Proposition 2. *Assume that a constant filter $\mathcal{F}$ is used and let S_k and T_k be the upper triangular and lower triangular matrices, respectively, of the dropped entries from U_{k+1} and L_{k+1} due to the filtering operation. Assume that the matrices L_k and U_k converge to $\bar{L}, \bar{U}$, respectively, and that $\bar{U}$ is nonsingular, Then,*

(i) *The sequence of matrices* S_k *converges to* $S = [\bar{L}^{-1}A]_{\triangledown} - \bar{U}$.

(ii) *The sequence of matrices* T_k *converges to* $T = [A\bar{U}^{-1}]_{\triangleleft} - \bar{L}$.

(iii) *The sequences of matrices* $(A - L_k U_{k+1})_{\triangleleft}$ *and* $(A - L_k U_k)_{\triangledown}$ *converge to* $[\bar{L}S]_{\triangledown}$ *and* $[T\bar{U}]_{\triangleleft}$, *respectively.*

Proof. To prove (i) note that under the assumptions, Equation (38) implies that $(L_k S_k)_{\triangledown}$ converges to $[A - \bar{L}\bar{U}]_{\triangledown}$. We now invoke Lemma 3 again and observe that:

$$[L_k^{-1}(L_k S_k)_{\triangledown}]_{\triangledown} = [S_k]_{\triangledown} = S_k$$

The left-hand side, and therefore also S_k, converges to the following limit

$$[\bar{L}^{-1}[A - \bar{L}\bar{U}]_{\triangledown}]_{\triangledown} = [\bar{L}^{-1}[A - \bar{L}\bar{U}]]_{\triangledown} = [\bar{L}^{-1}A]_{\triangledown} - \bar{U}.$$

The proof of (ii) is similar. The proof of (iii) is a direct consequence of relations (38–39). ∎

The convergence of the matrices of dropped entries may seem counter intuitive. However, note that these matrices represent the cumulative matrix of dropped entries. Our experience indicates that the algorithm often converges rapidly to a certain limit. With a trivial filter equal to a diagonal matrix, the process generally converges in one or two steps. We ran a few tests with finite difference discretizations of a Laplacien on an $n \times n \times n$ 3-D mesh, by taking the filter to be the original pattern of the matrix. We observed that the process converges in about N steps. In fact the norm $\|L_k - L_{k+1}\|_F + \|U_k - U_{k+1}\|_F$ drops abruptly at around n steps.

4. Practical Implementations

Two key tools must be exploited in any implementation of the methods discussed so far. One concerns basic sparse-sparse computations, i.e., computations involving products of sparse matrices with sparse vectors or matrices. The other key tool concerns effective ways of solving sparse triangular systems with sparse right-hand sides. Basic sparse-sparse computations have been covered in detail in the literature, see e.g., [13]. Techniques for solving sparse triangular systems with sparse right-hand sides are key ingredients of sparse direct solvers and so effective methods for this task have already been exploited. These are based on 'topological sorting', see, for example, [17, 31, 9]. Thus, an LU factorization can be implemented as a sequence of sparse triangular solves where the right-hand sides are sparse. Because the right-hand side is sparse, the solution is also sparse, and it is easy to find a sequence in which the unknowns must be determined by examining a tree structure issued from the sparsity pattern of both the right-hand side and the matrix. This well-known idea is summarized in the next subsection, and details can be found in, e.g., [17, 31, 14]. What is less undertood is how to adapt this idea to the context of approximate triangular solves. This is taken up in Section 4.2

4.1. Background: Solving sparse-sparse triangular systems

A sparse-sparse triangular linear system of equations is a linear system whose coefficient matrix as well as the right-hand side are sparse. It is common to model the tasks involved in a sparse

Figure 1. A small 10×10 sparse lower triangular matrix and its corresponding DAG

triangular system with the help of a a *Directed Acyclic Graph* (DAG) which represents the binary relation: *unknown i must be determined before unknown j*. An illustration is given in Figure 1.

It is useful to briefly recall a well-known technique used in sparse matrix methods for solving sparse-sparse triangular systems. A standard column version of a lower triangular system, such as the one shown in Figure 1 would first compute the first entry, x_1 of the solution. Then a multiple of the first column is subtracted from the right-hand side. The second unknown is determined and the process continues until all components have been calculated.

Assume now that the right-hand side is sparse. In this case, the solution vector will remain sparse during these column subtractions so at each step we just need to consider the next nonzero entry in the running right-hand-side b. The question is how to determine the next entry each time. Consider the example of the figure in the situation when $b = e_1$, the first column of the identity matrix. After x_1 is determined, the first column multiplied by x_1 is subtrated from $b = e_1$. This introduces 'fill-ins' in positions 2 and 4 of b. The next nonzero entry is now b_2 and we determine x_2. This introduces fill-ins in locations 5 and 6. At this point the next nonzero entry is b_4, so we determine x_4, etc.

Using matlab notation, the general algorithm is as follows (note that the first step copies the right-hand side onto the solution vector):

0. $x := b$, $j = 0$.
1. While $(j < N)$ Do
2. Determine $j :=$ index of next nonzero entry of x
3. Compute $x_j := x_j / L_{jj}$
4. Compute $x(j+1:N) := x(j+1:N) - x_j * L(j+1:N, j)$
5. End

The problem addressed by what is referred to a *topological sorting* is step 2 of the algorithm, namely finding the order in which to determine the unknowns. A topological sorting of the entries is obtained from the DAG. If $b = e_1$ then we perform a Depth-First Search traversal of the graph (visit subtrees, then visit current node) starting from the root at node 1. The topological sort of the nodes is then just a *reverse post-order traversal of the tree*, see [14].

For the above example, the post order traversal starting from node 1, is 5, 6, 2, 9, 7, 4, 1 so the topological order is 1, 4, 7, 9, 2, 6, 5. Note that not all the graph is traversed and that we get a valid order that is different from the natural one discussed above. For a general situation, where the right-hand side has more than one nonzero entry we will proceed in the same way but we now need to visit the tree from several roots in turn. For example, if the right-hand side for the above example is the vector $e_1 + e_3$, then we will do a traversal from node 1 and node 3. It is important to realize that the order is not important, i.e., starting the traversal from node 3 first and then node 1 or the other way around would both yield a valid order.

4.2. Approximate solutions of sparse-sparse triangular systems

So far we discussed only the exact solution case but we need to consider the situation of approximate solves since we are interested in approximate LU factorizations, not exact ones

as in sparse Gaussian Elimination. This particular situation does not seem to have been given much attention in the literature.

Suppose at first that $b = e_1$ again. In this situation we can just *truncate the DFS to a certain depth level*. This means that each node will get a level number inherited from its parent in the traversal, and when the level is higher than a certain maximum, denoted by '`MaxLev`' thereafter, the traversal is curtailed at that point. In the above example, a traversal with a `MaxLev` of 2 would yield 5, 6, 2, 7, 4, 1. This is easy to justify with the common model of level-of fill ILU factorizations. For model problems, the higher the level the smaller the entry being considered, and so for example the element x_9 is likely to be smaller than its parent x_7 which in turn is smaller than its parent x_4, etc.

A complication arises when we consider sparse right-hand sides with more than one nonzero entry. Consider again the situation when $b = e_1 + e_3$ in the above example. Then with respect to the traversal from node 1, x_7 has a level of 2, and x_9 a level of 3. With respect to the traversal from node 3, x_7 has a level of 1, and x_9 a level of 2. Which ones should we use? If we start the traversal from node 1 first, and if `MaxLev`=2, then 7 will be kept (lev=2) and 9 will be dropped (lev=3). Then when we will start the visit from node 3, the DFS traversal will see that node 7 was already visited and will ignore it.

We could resolve the issue by solving several systems with different right-hand sides, namely one for each nonzero entry, but this increases cost. A simpler solution, to prevent this is to *order the traversal from higher to lower numbered nodes* (in the example: first 3, then 1). This works simply because if $j > i$ then i cannot be a descendent of j.

4.3. A simplified iteration

Consider the situation when A is perturbed as $\tilde{A} = A + E$ where E is a very sparse matrix of small norm. Thus, we already have an approximate factorization $LU = A + R$, where R is the residual matrix in the approximation. After the perturbation we have $\tilde{A} = LU + E + R$. A natural idea suggested in [5] is to try to find a correction to L, U so that the residual norm remains the same as that of the original one. In other words we seek perturbations X_L, X_U to L, U respectively so that

$$(L + X_L)(U + X_U) = (A + E) + R$$

which leads to

$$X_L U + L X_U + X_L X_U - E = 0. \tag{40}$$

Note that this is the same equation as (4) with R simply replaced by E. We could approximately solve these equations by performing one iteration of the corresponding variant of (19–20):

$$U_{k+1} = U_k + [L_k^{-1} E]_{\Uparrow} \tag{41}$$

$$L_{k+1} = L_k + [E U_{k+1}^{-1}]_{\Downarrow}. \tag{42}$$

Observe that when E is lower triangular, for example $E = e_i e_j^T$ with $i > j$, then the first correction, (41), is vacuous, i.e., $U_{k+1} = U_k$, while the second correction gives

$$L_{k+1} = L_k + [E U_k^{-1}]_{\Downarrow}.$$

Approximating $[E U_k^{-1}]_{\Downarrow}$ by $[E(U)^{-1}]_{\Downarrow}$ yields the approximation used in [5]. The tests reported in [5] with this approach were for a specific application in CFD. Our experiments show that for

a general matrix and a general perturbation E then this very simple scheme does not improve the factorization to a satisfactory level.

An approach which we found to be more generally successful is one that is based on a simplification iterations (19–20). This simplification amounts to taking the *maximum depth level* `MaxLev` *in the triangular solves to be zero*, resulting essentially in replacing the triangular matrices by their diagonals. A few other minor changes are made to reduce cost. The algorithms is as follows where we assume that $A = L_0U_0 + R_0$ is the initial factorization.

Algorithm 3 (Simplified ITALU)

1. For $k := 0, 1, \ldots,$ Do:
2. Compute $R_k = A - L_kU_k$
3. Apply dropping to R_k.
4. Compute $U_{k+1} := U_k + (R_k)_{\nabla}$
5. Compute $L_{k+1} := L_k + (R_k)_{\nabla} * ((U_k))^{-1}$
6. End

Note that Line 4 substitutes for $U_{k+1} = U_k + (L_k^{-1}R_k)_{\nabla}$, i.e., it is the result of approximating L_k by the identity matrix. Similarly, Line 5 results from approximating the current U_k by its main diagonal. Note also that unlike Algorithm 2, the residual matrix R_k is only computed once per iteration. This contrasts with our original algorithm where R_k is recomputed after each update (L_k or U_k). Also dropping is applied to R_k and not to the corrections X_U and X_L.

If one iteration is performed, the cost of the above algorithm is quite modest. The computation of R_k and the 'pruning' step (steps 1 and 2) are basically negligible because they only require keeping track of entries that are dropped during the original LU factorization. The only cost is that of the logic to test whether an entry is to be dropped and the additional overhead in building the matrix R_k (memory allocation, pointer set-ups, etc). Steps 3 and 4 are equally inexpensive. They correspond to adding a hopefully very sparse matrix to the current factors L_k and U_k.

The remarkable aspect of the above Algorithm is that it works quite well in spite of its low cost, in the situation when the perturbation matrix is relatively small. A small number of iterations, e.g, 1 to 3, is usually sufficient to yield good improvements. An illustration is shown in the numerical experiments section, see Section 5.3.

5. Applications to linear and stationary problems

To illustrate the behavior of the incremental ILU, we consider in this section a few problems involving 3D-convection-diffusion like finite difference matrix. All experiments were performed in Matlab.

5.1. Alternating Lower-Upper corrections as a means to obtain an ILU

We will illustrate a method for extracting an ILU factorization by starting from the inaccurate SSOR defined below (or an ILU(0)) and then improving the factorization progressively by using one step of the procedure ITALU described in Section 3.

This experiment works on a stationary problem, specifically a matrix is generated from a convection-diffusion problem on a $15 \times 15 \times 10$ regular grid. This yields a problem of size $n = 2250$. We use finite difference discretization and generate a problem with constant convection terms in each direction. So, in each direction, the discretized operator yields a tridiagonal matrix

$$\text{tridiag}[-1+\alpha, \quad 2, \quad -1-\alpha].$$

In the first test α is taken equal to 0.1 and the matrix was shifted by 0.3 to make it indefinite. This shift introduces two negative eigenvalues and also two eigenvalues very close to zero as can seen from a run with MATLAB's eigs function:

$$\begin{array}{lll} \lambda_1 = -0.112843039478229, & \lambda_2 = -0.000397969025052, & \lambda_3 = 0.000397969025052, \\ \lambda_4 = 0.113638977528334, & \lambda_5 = 0.122450476821596 & \cdots \end{array}$$

In addition the condition number estimated by Matlab was 1.68e+05. The right-hand side of the system to solve is generated artificially as $b = A * ones(n, 1)$ in Matlab notation.

Figure 2 shows the performance of GMRES with 4 different preconditioners. The first preconditioner is SSOR with $\omega = 1$. In fact this is simply the factorization $A \approx L_0 U_0$ with $L_0 = I + A_{\mathbb{L}}(A)^{-1}$ and $U_0 = A_{\mathbb{U}}$. The second preconditioner takes these two factors and generates an improved factorization $A \approx L_1 U_1$ by performing one iteration of the ITALU algorithm. Dropping is performed in this procedure as follows. First a relative tolerance is set to `droptol` $= 0.2$, the effect of which is to ignore entries in any generated column that is less than `droptol` times the norm of the original row. In addition, as a precaution against excessive fill-in, only the largest `lfil`=10 entries are kept (however in this example, this second criterion does not seem to have been triggered). The results, in the form of residual norm versus iteration count, are shown on the left panel of Figure 2.

In the last run, corresponding to ITALU(3), the resulting preconditioner yielded a convergence behavior that is identical with that obtained from the previous one (i.e., from ITALU(2)). Therefore, we changed the parameters to show one more run, namely one corresponding to `droptol` $= 0.1$ and `lfil` $= 20$ (starting again from the preconditioner obtained in the previous run, i.e., the one labelled ITALU(2) in the figure). The numbers shown at the end of each curve (3.64, 1.97, 1.78, 1) are the "fill-factors" for each preconditioner, i.e., the ratios of the number of nonzero entries in $L+U$ over that of A. These measure the amount of memory required to store the preconditioner and give a rough measure of the computational effort to construct it.

In a second test, we repeated exactly the same experiment with a problem of the same size but with different coefficients: we took $\alpha = 0.2$ and then shifted the matrix by the diagonal -0.5 I to make the problem more indefinite. In this case SSOR has trouble converging. The results are shown on the right panel of Figure 2. The smallest eigenvalues as computed by the function eigs are now

$$\begin{array}{lll} \lambda_1 = -0.2240921324 & \lambda_2 = \lambda_3 = -.1125800958 & \lambda_4 = -0.0010680592 \\ \lambda_5 = \ .0076089054 & \lambda_6 = \lambda_7 = 0.0685056254 & \lambda_8 = \ .1191209420 \ ... \end{array}$$

and Matlab's condition number estimator yields $condest(A) \approx 4.03E+05$.

Recall that the test corresponding to ITALU(3) used `droptol`=0.1 and `lfil`=20 instead of `droptol`=0.2 and `lfil`=10. The fill-factor is now substantially larger for this last test, but this results in a much faster convergence.

Figure 2. Performance of GMRES(30) for a convection-diffusion problem with the SSOR and 3 levels of improvements by the ITALU algorithm. Left and right plot are for two different problems.

5.2. Descent-type methods for computing an ILU

The Frobenius-based method MERLU described in Section 2.2 will now be illustrated on the second example seen in the previous section, i.e., the Convection-Diffusion problem, with the parameter $\alpha = 0.2$ for the convection term and the shift -0.5.

The following test is similar to the one described above. First GMRES(30) is attempted with the SSOR preconditioner. This yields the first convergence curve of the left side of Figure 3. Then two improved preconditioners are generated by Algorithm 1. The same dropping parameters are applied in Lines 5 and 10 as in the previous example. We took `droptol` = 0.2 and `lfil`=10. As before the number shown at the end of each curve corresponds to the fill-factor for each case. The last curve shows the performance of an algorithm which is a modification of Algorithm 1 in which the dropping is done differently. In this variation (labeled MERLU1), the dropping in Lines 5 and 10 is removed. Instead, we drop directly from the factors U and L after each update. From a practical point of view this is more interesting because fill-in is better controlled. From a theoretical point of view, we loose the guarantee that the factors generated will have a diminishing residual norm $\|A - LU\|_F$. The last curve in the figure shows this result with the preconditioner obtained from 4 steps of this procedure starting with the SSOR factors as the initial guess. The dropping parameters are again `droptol` = 0.2, `lfil` = 10.

We repeated the experiment with less restrictive fill-in parameter by changing the value of `lfil` to 5 instead of 10. This means that fewer entries will be kept each time. The results are shown on the right side of Figure 3. The alternative procedure MERLU1 converges slowly but requires very little fill-in (a fill ratio of 1.31). In fact the last few iterations of the iterative LU procedure to compute the LU factors are not very effective here because we keep too few entries at each iteration. As a result an almost identical convergence curve is obtained if the number of iterations is reduced from 4 to 2. To remedy this, we tested yet another variant which allowed more fill in after each outer iteration. Specifically, we simply increase `lfil` by 2 after each iterative LU computation. The resulting method is labeled MERLU2 in the plot. As can be seen this variant achieves good convergence relative to the fill-in allowed.

Another set of experiments performed to show an interesting application of the incremental approach is one in which the ILU factorization starts from a basic one such as an SSOR

Figure 3. Performance of GMRES(30) for a convection-diffusion problem with the SSOR and a few levels of improvements using the Frobenius-based approach. The test matrix is the same as the second one of the previous section. The right plot shows tests with different dropping strategies.

preconditioner, and then is gradually improved at the occasion of each GMRES restart. We do not report these experiments here.

5.3. Use of Topological Sort and Simplified ITALU

His experiment focuses on the issue of efficient implementations, see Section 4.1. Though our current implementations are in matlab, it is possible to get an idea of the cost of the techniques by means of a rough operation count.

The experimental setting is as follows. We generate a problem similar to the ones of the previous section taking this time $n_x = n_y = 70, n_z = 1$, $\alpha = 0.05$ and the resulting matrix is shifted by $\sigma = -0.01$. The matrix size is now $n = 4,900$. We use the same artificial right-hand side as before, namely, $b = A * ones(n, 1)$.

Six experiments are run on this example. First we solve the system with GMRES(30)-ILUT using matlab's `luinc` with a `droptol` of 0.1. The convergence is slow and the wanted reduction of the residual norm by a factor of 1.e-08 was not reached in 200 steps.

We then perturb the matrix as follows. We generate the following perturbation using matlab commands:

```
z = zeros(n,1); v = z;
k = fix(n/20);    k1 = fix(k/2);
z(k:k:n) = (0.15/n)*(k:k:n);
v(k1:k:n) = (0.15/n)*(k1:k:n);
E = sparse(v) * sparse(z)';
```

This matrix E is added to A to produce the perturbed matrix B. This is a rank-one perturbation to A, and so there exist more effective ways to solve the new linear system, via the Sherman-Morisson formula[19] but we selected this form of perturbing A only for the sake of reproducibility. Perturbations with other randomly generated matrices yielded similar results. We now want to solve the linear systems $Bx = b$, with the same right-hand side b as before. The next test is to just try using the same preconditioner to solve the new linear

Figure 4. Performance of GMRES(30) for a convection-diffusion problem with one or two levels of improvements of the ITALU algorithm (left side) and the simplified version described by Alg. 3.

system. This usually works reasonably well when the perturbation is small and / or of small rank. In this case however, the convergence is quite slow, substantially slower than with the original system.

We then tested the performance of ITALU with the topological sorting algorithm discussed in Section 4.1. We used a Maximum Level of 3. The left side of Figure 4 shows the performance of GMRES(30) with the 4 tests based on topological sorting. These runs correspond to: 1) Original problem solved with GMRES(30) using the preconditioner ILUT(0.1) on the original problem. 2) Performance of the same preconditioner-accelerator pair on the perturbed problem. 3) Performance of the corrected preconditioner using `MaxLev`=3 in the topological sort. One iteration is made. 4) Performance of the corrected preconditioner using `MaxLev`=3 in the topological sort. Two correction iterations made.

The first two curves seen on the right side of the same figure show the same result as those on the right side and they are reproduced here to facilitate comparisons. Note that the small differences are caused by the fact that the initial guesses are random vectors. The curves 3) and 4) are replaced by their analogues using the simplified version of ITALU as represented by Algorithm 3. The numbers at the end of each curve indicate the fill-factors for each case, see Section 5.2.

While the matlab codes used in the first experiment is expensive relative to the C-based `luinc` it is clear that the simplified version should be very inexpensive. To give an idea, the time to construct the initial preconditioner for this example by `luinc` is 0.08 sec. The time to compute the first correction is 0.0493 and that for the second correction 0.08 (all seconds). It is to be stressed that the `luinc` code of matlab is optimized whereas the computations of the corrections only used simple-minded matlab functions (e.g., `triu,..`).

6. An application in an evolutive computational fluid dynamics problem

This section discusses an application of the methods proposed in this paper to the numerical simulation of the variable density incompressible Navier-Stokes system given on a domain $\Omega \subset \mathbb{R}^2$. These equations model the motion of two immiscible fluids with varying density, evolving in a domain Ω. This is the case of the Rayleigh–Taylor instabilities or the water bubble falling in air – see [7]. These phenomena vary locally both in the physical domain and in time and the corresponding operators are time dependent.

6.1. Governing equations and discrete problems

The variable density incompressible Navier-Stokes system in $\Omega \subset \mathbb{R}^2$ can be written as:

$$\partial_t \rho + \mathrm{div}_{\mathbf{x}}(\rho \mathbf{u}) = 0, \tag{43}$$

$$\rho\big(\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla_{\mathbf{x}})\mathbf{u}\big) + \nabla_{\mathbf{x}} p - \mu \Delta_{\mathbf{x}} \mathbf{u} = \mathbf{f}, \tag{44}$$

$$\mathrm{div}_{\mathbf{x}} \mathbf{u} = 0. \tag{45}$$

Here $\rho(t, \mathbf{x}) \geq 0$ stands for the density of a viscous fluid whose velocity field is $\mathbf{u}(t, \mathbf{x}) \in \mathbb{R}^2$.

The description of the external force is embodied into the right hand side $\mathbf{f}(t,\mathbf{x})$ of (44) and $\mu > 0$ is the viscosity of the fluid. The unknowns depend both on time $t \geq 0$ and position $\mathbf{x} \in \Omega \subset \mathbb{R}^2$. The third unknown of the problem is the pressure $p(t,\mathbf{x}) \in \mathbb{R}$; it can be seen as a Lagrange multiplier associated to the incompressibility constraint (45). The problem is well posed when the equations are completed by initial and boundary conditions.

In [7], different numerical methods have been proposed for the transport equation (43) and for evaluating the evolution of the velocity driven by (44–45). To be more specific, thanks to a time-splitting (the "Strang splitting" [30]), it is possible to solve (43) for a given velocity by using a Finite Volume approach (see e.g. [16, 27]) which is efficient when dealing with a pure convection equation and then, compute the divergence free solution of the momentum equation (44) by exploiting the advantages of Finite Element (FE) methods (see e.g. [18, 15]).

For this, the domain Ω is approximated by a computational domain Ω_h, discretized by a conforming and isotropic set of triangles $\mathcal{T}_h$, with mesh-size h. The FE spaces $V_h \subset (H_0^1(\Omega_h))^2$ for the velocity $\mathbf{u}_h$ and $Q_h \subset L_0^2(\Omega_h)$ for the pressure p_h must verify the "inf-sup" or LBB condition. In what follows we shall use the classical $\mathbb{P}_2/\mathbb{P}_1$ elements in order to obtain a "saddle point problem" corresponding to the momentum equation (44–45). We also need to define a suitable discrete approximation of the (given) function ρ, compatible with the discretization of the velocity and the pressure. We use $\mathbb{P}_1$ elements for the natural choice $\rho_h \in \bar{Q}_h \subset L^2(\Omega_h)$.

Let us denote by Δt the time step and $t^n = n\,\Delta t$, $n \geq 0$. For the time discretization of (44-45), we use a classical backward difference method for the time derivative and a semi-implicit linearization scheme to treat the nonlinear convection term in the momentum equation (44). This scheme, already used in [22], has a second-order accuracy in time. Specifically, given the approximations $\mathbf{u}^n, \mathbf{u}^{n-1}$ and ρ^* ($\rho^* = \rho^n$ or ρ^{n+1} depending on the time step of the Strang splitting), at time t^{n+1} we compute $(\mathbf{u}^{n+1}, p^{n+1})$ by solving

$$\rho^* \left(\frac{3\mathbf{u}^{n+1} - 4\mathbf{u}^n + \mathbf{u}^{n-1}}{2\Delta t} + (\bar{\mathbf{u}}^{n+1} \cdot \nabla_{\mathbf{x}})\mathbf{u}^{n+1} \right) - \mu \Delta_{\mathbf{x}} \mathbf{u}^{n+1} + \nabla_{\mathbf{x}} p^{n+1} = \mathbf{f}^{n+1}, \quad (46)$$

$$\mathrm{div}_{\mathbf{x}} \mathbf{u}^{n+1} = 0. \quad (47)$$

Here, $\bar{\mathbf{u}}^{n+1} = 2\mathbf{u}^n - \mathbf{u}^{n-1}$ is the linear second-order extrapolation of the velocity field at t^{n+1}.

The algebraic version of discrete variational form of (46-47) may be written in the following saddle point problem:

$$\begin{pmatrix} \mathbf{A}_{n+1} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}, \quad (48)$$

where $\mathbf{A}_{n+1}$ is a nonsymmetric matrix associated with the convection-diffusion term, $\mathbf{B}$ is the matrix associated with the divergence operator, and the vectors U, P, F correspond, respectively, to the unknowns $\mathbf{u}_h^{n+1}, p_h^{n+1}$ and to the given function $\mathbf{f}_h^{n+1}$ plus other explicit terms depending on the velocity. Here, $\mathbf{A}_{n+1}$ changes from one time step to the next, due to the evolution of the density and to the linearization of the convective term. Specifically, we have $\mathbf{A}_{n+1} = \frac{3}{2\Delta t}\mathbf{M} + \frac{1}{Re}\mathbf{L} + \mathbf{D}$, where $\mathbf{M}$ is the velocity mass matrix depending on ρ_h^*, $\mathbf{L}$ is the Laplacian matrix for the velocity and the matrix $\mathbf{D}$ corresponds to the convective term and depends on ρ_h^* and $\bar{\mathbf{u}}_h^{n+1}$. Typically, the matrices $\mathbf{M}$ and $\mathbf{D}$ change at each time step and are not symmetric. Finally, recall that the Reynolds number Re is proportional to the size of the domain, a reference density and a reference velocity and inversely proportional to the dynamic viscosity μ.

In the case of homogeneous density $\rho(t,\mathbf{x}) = \overline{\rho},\ \forall t > 0$, the saddle point problem (48) has been studied in many papers, see e.g. [8] and references therein. In these references block diagonal or block triangular preconditioning strategies are advocated for the system matrix of (48). The eigenvalue analysis in these references shows that the eigenvalues of the preconditioned system are bounded independently of the mesh size h of the underlying grid $\mathcal{T}_h$ and numerical tests show that the convergence rates for GMRES iterations deteriorates roughly like the Reynolds number Re.

The problem considered is to solve the consecutive systems (48) by taking advantage of earlier systems. In this section, we study the influence of the alternating lower-upper correction, Algorithm 2, as preconditioner of the nonsymmetric matrix $\mathbf{A}_{n+1}$, in the case of a block triangular preconditioner

$$\begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{B}^T \\ \mathbf{0} & \widehat{\mathbf{S}} \end{pmatrix} \text{ or } \begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{0} \\ \mathbf{B} & \widehat{\mathbf{S}} \end{pmatrix},$$

applied to the saddle point problem, where $\widehat{\mathbf{A}}$ is an approximation of the nonsymmetric matrix $\mathbf{A}_{n+1}$ and $\widehat{\mathbf{S}}$ is an approximation of the pressure Schur complement $\mathbf{B}\mathbf{A}_{n+1}^{-1}\mathbf{B}^T$. Since FGMRES, the flexible variant of GMRES, allows variations in the preconditioner, it is used as the accelerator in our implementation. In FGMRES, preconditioning is applied to the right:

$$\begin{pmatrix} \mathbf{A}_{n+1} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{B}^T \\ \mathbf{0} & \widehat{\mathbf{S}} \end{pmatrix}^{-1} \begin{pmatrix} W_U \\ W_P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}, \text{ with } \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{B}^T \\ \mathbf{0} & \widehat{\mathbf{S}} \end{pmatrix}^{-1} \begin{pmatrix} W_U \\ W_P \end{pmatrix}.$$

We focus our study only on the preconditioner $\widehat{\mathbf{A}}$. For the approximation of the pressure Schur complement $\mathbf{B}\mathbf{A}_{n+1}^{-1}\mathbf{B}^T$, we choose two classical preconditioners: the pressure-mass preconditioner [29] $\widehat{\mathbf{S}}^{-1} = \mathbf{M}_p^{-1}$ and the Cahouet-Chabard preconditioner [6] $\widehat{\mathbf{S}}^{-1} = \frac{2\Delta t}{3}\mathbf{M}_p^{-1} + Re\mathbf{L}_p^{-1}$, where $\mathbf{M}_p$ is the mass matrix for the pressure and $\mathbf{L}_p$ is the discretization of the Laplace operator for the pressure.

6.2. The ITALU algorithm for saddle point problems

The efficiency of a preconditioner for the saddle point problem (48) can be tested numerically by verifying that the convergence rates for FGMRES iterations depend only mildly on the Reynolds number and, most importantly, that the number of iterations does not grow as the mesh size is reduced.

We want to evaluate the ability of the Algorithm 2 to recover analytical solution, checking the rates of convergence and comparing the mean number of iterations required by standard ILUT preconditioning (using Matlab `luinc` function with `droptol` $= 10^{-5}$) with those required by the ITALU algorithm. In particular, we use the same `droptol` and we increase the `lfil` and `itlu` parameters when h decreases. Here, `itlu` is the number of iterations performed by the ITALU algorithm. We check for each test the rates of convergence of the velocity and pressure considering the example given in [7]:

$$\left\{ \begin{array}{rcl} \rho_{\text{ex}}(t,x,y) & = & 2 + x\cos(\sin t) + y\sin(\sin t), \\ \mathbf{u}_{\text{ex}}(t,x,y) & = & \begin{pmatrix} -y\cos t \\ x\cos t \end{pmatrix}, \\ p_{\text{ex}}(t,x,y) & = & \sin x \sin y \sin t. \end{array} \right. \tag{49}$$

Figure 5. Mean number of iterations required by FGMRES(30) for the saddle point problem using pressure-mass preconditioner for the Schur complement, for different values of h and $Re = 1$.

The fields $\rho_{\rm ex}(t,x,y)$ and $\mathbf{u}_{\rm ex}(t,x,y)$ satisfy the mass conservation equation (43) identically and $\mathbf{u}_{\rm ex}(t,x,y)$ is solenoidal. The momentum equation (44) is satisfied with the body force:

$$\mathbf{f}_{\rm ex}(t,x,y) = \begin{pmatrix} (y\sin t - x\cos^2 t)\rho_{\rm ex}(t,x,y) + \cos x \sin y \sin t \\ -(x\sin t + y\cos^2 t)\rho_{\rm ex}(t,x,y) + \sin x \cos y \sin t \end{pmatrix}. \tag{50}$$

Computations are performed for $0 \le t \le 0.5$ on the square $[-1,1]^2$, using a structured mesh $\mathcal{T}_h$. The convergence results are displayed with respect to h and for different time steps Δt. Table I shows the mesh size, the corresponding dimensions of vectors U and P and the number of nonzero entries in the matrix $\mathbf{A}_{n+1}$, which is block-diagonal.

Table I. Dimensions of the problem with respect to the mesh size.

h	$size(U)$	$size(P)$	$nnz(\mathbf{A}_{n+1})$
$h_1 = 0.0312$	1832	289	20578
$h_2 = 0.0156$	7938	1089	88162
$h_3 = 0.0078$	32258	4225	364642

For the operation with the matrix $\widehat{\mathbf{S}}^{-1}$, we always use the L and U factors obtained from Matlab's ILUT (the `luinc` procedure), with the `droptol` set to `droptol=` 10^{-3} and no pivoting.

The results in Figure 5 correspond to the pressure-mass preconditioner $\widehat{\mathbf{S}}^{-1} = \mathbf{M}_p^{-1}$ for the saddle point problem, when $Re = 1$ and $h = h_1, h_2, h_3$. In order to compare the results of a standard approach and the new procedure presented in section 3, two algorithms are used for operations with the matrix $\widehat{\mathbf{A}}$: the first one is the standard ILU preconditioner of Matlab `luinc`, by setting `droptol` $= 10^{-5}$ and no pivoting. The second one is the ITALU procedure with the same `droptol`. In this figure and the next two ones, the mean value of the fill-ratio of the preconditioner $\widehat{\mathbf{A}}$ is displayed for each case. Each of these plots shows iteration counts (vertical axis) versus $-\log_{10}(\Delta t)$ (horizontal axis). The left plots show performances of the Matlab `luinc`(drop) preconditioning and the right plots those of the ITALU procedure.

It seems obvious that the parameters `lfil` and `itlu` should not be fixed. They should vary according to the mesh size h. In particular, we set `lfil` = 20,25,30 (or 40) and `itlu` = 3,4,5

Figure 6. Mean number of iterations required by FGMRES(30) for the saddle point problem using the Cahouet-Chabard preconditioner for the Schur complement, for different values of h and $Re = 1$.

Figure 7. Mean number of iterations required by FGMRES(30) for the saddle point problem using the Cahouet-Chabard preconditioner for the Schur complement, for different values of h and $Re = 1000$.

respectively, when the mesh size h decreases. Smaller values of `lfil` or `itlu`, result in a greater number of FGMRES iterations on average. The results are shown on the right side of Figure 5. Near $h = h_3$ we indicate the `lfil` used. As expected, we can observe that the `lfil` parameter is very important to obtain a rate of convergence independent of the mesh size.

Using the same values for h, we test the Cahouet-Chabard preconditioner $\widehat{\mathbf{S}}^{-1} = \frac{2\Delta t}{3}\mathbf{M}_p^{-1} + Re\mathbf{L}_p^{-1}$, for the saddle point problem, when $Re = 1$ and $Re = 1000$. This choice is necessary because for higher Reynolds numbers the performances of the pressure-mass preconditioner for the Schur complement are not satisfactory.

The results in Figure 6 (resp. Figure 7) correspond to $Re = 1$ (resp. $Re = 1000$). The results in the left side of these figures correspond to the standard ILUT preconditioner and those in the right side to the ITALU preconditioner. If $Re = 1$, we take the same parameters `droptol`, `lfil` and `itlu` as in the previous test case. As in the preceding case, the importance of the `lfil` parameter can be observed for small values of h. With higher Reynolds numbers, we must increase the parameters `lfil` and `itlu` in the ITALU preconditioner. In particular, we set `lfil` = 30,40,50 (or 70) and `itlu` = 3,4,5 respectively, when the mesh size h decreases. In this case, choosing higher `lfil` value does not reduce the mean number of iterations of

FGMRES; the only consequence is an increased fill-in of $\widehat{\mathbf{A}}$.

Overall, we stress that the fill-in of $\widehat{\mathbf{A}}$ is reduced in a significant way (up to 40%) when considering the ITALU procedure presented in Section 3 with respect to the classical approach, in particular when Re is very large. This crucial point will become very important when considering large problems for which memory costs will cause severe limitations.

However, to capture correctly the dynamics with higher Reynolds numbers, only small time steps Δt must be chosen and in this case the performances of FGMRES are equivalent.

We also underline that, when we proceed through the sequence of problems, the number of iterations of FGMRES does not change very much with respect to the mean iteration number plotted in Figures 5, 6 and 7 . Indeed, for both ITALU and `luinc` algorithms, the FGMRES iterations fluctuate between approximatively +10 % of the mean iteration number, at the beginning of the time stepping iterations, and decrease until -10% at the end of simulations.

7. Conclusion

Though much is known about finding effective preconditioners to solve general sparse linear systems which arise in real-life applications, little has been done so far to address the issue of updating such preconditioners. This paper proposed a few possible practical approaches. One option is to utilize techniques based on sparse approximate inverses. These methods can yield very inexpensive ways to correct LU factors. They rely on products of sparse matrices with sparse vectors which have been exploited in the literature, see, e.g., [13].

This paper also considered an alternating procedure, named ITALU, which corrects the factors L and U alternatingly. Experiments show that the method can perform quite well. The main ingredient in the construction of ITALU preconditioners is the solution of a sparse triangular system with a sparse right-hand side matrix. Computations of this type constitute the innermost kernels of sparse direct solvers, see, e.g., [14, 17, 31]. This paper has shown how to exploit the idea of levels in the graph traversal to adapt these techniques for the case when only an approximate solution is desired. At one extreme one can obtain a very inexpensive and surprisingly effective update of the ILU factorization by solving these systems in a very approximate manner. The corresponding technique, called the simplified ITALU, yielded nice improvements on the examples we tested. One can make the corrections as inexpensive as desired, by adjusting the depth levels in the sparse triangular solve. At the other extreme, one can obtain full-fledged methods for building accurate factorizations starting from, say, a diagonal or an SSOR factorization.

The possibility of computing updates of ILU factorizations inexpensively opens up many interesting avenues. Many applications in CFD lead naturally to small variations in the matrix. A feature that is common to many physical applications is that the varying parts or the 'rough' parts of the domain are usually limited in size. In examples involving the interaction of two fluids or a solid body and a fluid, the change in the coefficient matrix is localized around the physical interfaces. In this situation, the largest entries of the residual matrix R_k will be localized too and the costs of the updates will be quite small. It is also possible to devise strategies whereby an ILU factorization is built with a certain uniform tolerance for the error $\|A - LU\|_F$ and then the factors are improved locally only where the domain is likely to cause difficulties. In this way one can improve the quality of the preconditioner only in areas where it is more likely that a higher accuracy is required. These are but a few of the topics which

are left for future studies.

REFERENCES

1. O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, New York, 1994.
2. M. Benzi, R. Kouhia, and Tůma, *Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics*, Comput. Methods Appl. Mech. Engrg., 190 (2001), pp. 6533–6554.
3. M. Benzi, C. D. Meyer, and Tůma, *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM Journal on Scientific Computing, 17 (1996), pp. 1135–1149.
4. M. Benzi and M. Tůma, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM Journal on Scientific Computing, 19 (1998), pp. 968–994.
5. P. Birken, J. D. Tebbens, and M. Meister, Andreas andTuma, *Preconditioner updates applied to CFD model problems*, Applied Numerical Mathematics, (2008). In press.
6. J. Cahouet and J.-P. Chabard, *Some fast 3D finite element solvers for the generalized Stokes problem.*, Int. J. Numer. Methods Fluids, 8 (1988), pp. 865–895.
7. C. Calgaro, E. Creusé, and T. Goudon, *An hybrid finite volume-finite element method for variable density incompressible flows*, J. Comput. Phys., 227 (2008), pp. 4671–4696.
8. C. Calgaro, P. Deuring, and D. Jennequin, *A preconditioner for generalized saddle-point problems: Application to 3d stationary Navier-Stokes equations*, Numer. Methods Partial Differential Equations, 22 (2006), pp. 1289–1313.
9. S. M. Chan and V. Brandwajn, *Partial matrix refactorization*, IEEE trans. Power Systems, (1986), pp. 193–199.
10. J.-P. Chehab, *Matrix differential equations and inverse preconditioners*, Computational and Applied Mathematics, 26 (2007), pp. 95–128.
11. J.-P. Chehab and J. Laminie, *Differential equations and solution of linear systems*, Numerical Algorithms, 40 (2005), pp. 103–124.
12. E. Chow and Y. Saad, *Approximate inverse techniques for block-partitioned matrices*, SIAM Journal on Scientific Computing, 18 (1997), pp. 1657–1675.
13. ———, *Approximate inverse preconditioners via sparse-sparse iterations*, SIAM Journal on Scientific Computing, 19 (1998), pp. 995–1023.
14. T. A. Davis, *Direct methods for sparse linear systems*, SIAM, Philadelphia, PA, 2006.
15. A. Ern and J.-L. Guermond, *Éléments finis: théorie, applications, mise en œuvre*, vol. 36 of Mathématiques & Applications (Berlin) [Mathematics & Applications], Springer-Verlag, Berlin, 2002.
16. R. Eymard, T. Gallouët, and R. Herbin, *Finite volume methods*, in Handbook of numerical analysis, Vol. VII, Handb. Numer. Anal., VII, North-Holland, Amsterdam, 2000, pp. 713–1020.
17. J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to artithmetic operations*, SIAM Journal on Scientific Computing, 9 (1988), pp. 862–874.
18. V. Girault and P.-A. Raviart, *Finite element methods for Navier-Stokes equations*, vol. 5 of Springer Series in Computational Mathematics, Springer-Verlag, Berlin, 1986. Theory and algorithms.
19. G. H. Golub and C. F. V. Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 3rd ed., 1996.
20. M. Grote and H. D. Simon, *Parallel preconditioning and approximate inverses on the connection machine*, in Parallel Processing for Scientific Computing – vol. 2, R. F. Sincovec, D. E. Keyes, L. R. Petzold, and D. A. Reed, eds., SIAM, 1992, pp. 519–523.
21. M. J. Grote and T. Huckle, *Parallel preconditionings with sparse approximate inverses*, SIAM Journal on Scientific Computing, 18 (1997), pp. 838–853.
22. J.-L. Guermond and L. Quartapelle, *A projection FEM for variable density incompressible flows*, J. Comput. Phys., 165 (2000), pp. 167–188.
23. R. M. Holland, A. J. Wathen, and G. Shaw, *Sparse approximate inverses and target matrices*, SIAM J. Sci. Comput., 26 (2005), pp. 1000–1011.
24. L. Y. Kolotilina and A. Y. Yeremin, *On a family of two-level preconditionings of the incomplete block factorization type*, Soviet Journal of Numerical Analysis and Mathematical Modeling, 1 (1986), pp. 293–320.
25. ———, *Factorized sparse approximate inverse preconditionings I.Theory.*, SIAM Journal on Matrix Analysis and Applications, 14 (1993), pp. 45–58.
26. L. Y. Kolotinina and A. Y. Yeremin, *Incomplete block factorizations as preconditioners for sparse spd matrices*, in Recent advances in iterative methods, IMA Volunes in Math. Appl., 60, New York, 1994, Springer.

27. R. J. LeVeque, *Finite volume methods for hyperbolic problems*, Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge, 2002.
28. Y. Saad, *Iterative Methods for Sparse Linear Systems, 2nd edition*, SIAM, Philadelpha, PA, 2003.
29. D. Silvester, H. Elman, D. Kay, and A. Wathen, *Efficient preconditioning of the linearized Navier-Stokes equations for incompressible flow.*, Journal of Computational and Applied Mathematics, 128 (2001), pp. 261–279.
30. G. Strang, *On the construction and comparison of difference schemes*, SIAM J. Numer. Anal., 5 (1968), pp. 506–517.
31. W. F. Tinney, V. Brandwajn, and S. M. Chan, *Sparse vector methods*, IEEE trans. Power Apparatus and Systems, (1985), pp. 295–301.